ClojureScript One

Tuesday, February 14, 2012
8:54 AM

We think that ClojureScript is amazing and that it provides endless possibilities for making web development much more fun. To realize these possibilities, we need your help. Someone has to do the work to turn the possible into reality.

We think that the best way to convince you to help us is to allow you to experience the wonder of ClojureScript development for yourself. The purpose of this project is to help you get started with ClojureScript as smoothly and quickly as possible and to set you up with an excellent working environment.

Getting started means many things. We often forget how much we need to know to be comfortable in a programming environment. How do we organize code? How do we test? What is the most productive workflow? How do I deploy my application? We will attempt to answer all of these questions and provide working examples.

The project includes a working sample application, useful tools, and libraries in various states of development. Many of the libraries in the project will eventually mature and become their own projects. The process of creating good libraries and frameworks takes time and experience. Instead of waiting until they are finished, we thought it would be better to show you now. Once you see the potential, we know that you will want to join in and help.

Why is it called "ClojureScript One"?

ClojureScript is well suited for writing client-service applications which must run reliably for long periods of time within a single page.

The project is named ClojureScript One because it is the first step in learning to write these applications which are also written in one language on both the client and the server. You deploy a single JavaScript file and can think in terms of one process and one paradigm. There is one data format for representing data both within an application and while being transmitted across the network. Finally, as you work with the included tools you will start to notice that the line between tool and application will blur and the two will appear to become one.

Apart from Clojure and ClojureScript, ClojureScript One is built on top of Ring, Compojure, Enlive and Domina. The code for every other feature is included in this repository. If you go through and understand all of this code you will learn a lot about ClojureScript. It is our hope that learning will pay off with new and even better ideas than those presented here.

Getting Started

Tuesday, February 14, 2012
8:53 AM

To install ClojureScript One, install leiningen, clone ClojureScript One and, from within the cloned directory, run `lein bootstrap`.

```
git clone https://github.com/brentonashworth/one.git
cd one
lein bootstrap
lein repl
(go)
```

The commands above will install everything, start the server, and start a browser that opens to http://localhost:8080.

Getting an exception when you run `lein repl`?

If you get an exception when you run `lein repl`, try this workaround.

Clojure REPL

The command `lein repl` will start a properly classpathed Clojure Read-eval-print-loop (REPL). To start the development server, you can then type `(dev-server)` at the prompt which appears.

To interact with the running ClojureScript application, you will need to start a ClojureScript REPL. To start the development server, a ClojureScript REPL, and open a browser to http://localhost:8080, type `(go)` instead of `(dev-server)`.

About the ClojureScript REPL

A ClojureScript REPL allows you to evaluate forms in the running ClojureScript application. Starting a working ClojureScript REPL is a two step process. You first need to start the REPL process and then you need to connect it to the browser. To connect, visit http://localhost:8080 and click the **Development** button. If you are already on this page then you will need to refresh it. Each time you start a ClojureScript REPL you will need to either go to or refresh this page.

The ClojureScript REPL will only work when running the development version of the application. Neither the design nor the production view connects to the REPL.

Testing your connection

With your browser window in view, evaluate the following forms to ensure that you are connected and to experience some of the joy of ClojureScript.

```
(js/alert "hello")
(in-ns 'one.sample.view)
(set-text! (by-id "header") "The Form")
(set-styles! (by-id "header") {:color "#FF0000" :font-size "138px"})
```

Entering `:cljs/quit` in the ClojureScript REPL will stop this REPL and put you back in the Clojure REPL. To re-enter the ClojureScript REPL, simply type `(cljs-repl)`. In this way, one can quickly switch between the Clojure and ClojureScript REPL when working on the client and server portions of an application. Remember that you will need to refresh the development view in the browser every time you start a new ClojureScript REPL.

Workflow

Tuesday, February 14, 2012
8:53 AM

When working with ClojureScript, having a good workflow is essential. The first thing to know about workflow is that you should be working from the REPL as much as possible. In fact, if you are not spending most of your time in the REPL, you're doing it wrong.

Using the REPL as the main way to deliver code to the browser means never having to refresh the page. One could theoretically build an entire application without a single page refresh. If you find yourself refreshing the page after every change you make, you're doing it wrong. What is this, 2009?

The documentation in this wiki will often encourage you to start a REPL and follow along. We suggest that you follow this advice as much as possible. The best way to understand the sample application and to learn how to work with any ClojureScript application is to follow along while reading through this wiki.

As you begin working on your own applications, it is important to have a development tool that allows you to edit source code and easily evaluate it in a REPL. Emacs is great for this. If you're not into Emacs, there are other options.

Design -> Development -> Production

ClojureScript One is a development tool which also hosts the application being developed. By default, the tool gives you three views of your application: Design, Development and Production.

Before you begin doing anything with ClojureScript One, you should first have a plan for what you are going to build. The plan should, at least, include the views that you would like to render and the events that cause you to transition between views.

With this plan in place, your first stop should be the **Design** view. In this view you use HTML, CSS and Images to create static views for your application. You can treat these as templates which may be dynamically manipulated and rendered at runtime.

Having this separate view of static resources is very helpful for both developers and designers.

Once the static views have been created, the **Development** area allows you to run, interact with and build your application. The JavaScript running on this page is produced by the ClojureScript compiler alone. The application is not optimized or minified. Each input ClojureScript file becomes one JavaScript file. This makes debugging a little easier.

ClojureScript supports another kind of compilation which performs optimizations, dead-code elimination and minification. This is called "advanced compilation" and should be checked periodically. The **Production** view hosts the advanced compiled version of the application. Visiting this view will cause your application to be compiled. Even though ClojureScript supports advanced compilation, it is still possible to do things that will break it. This check should be performed regularly.

When looking at the advanced compiled version of an application, notice that all of the JavaScript lives in one minified JavaScript file. You will find the compiled JavaScript files in the `public/javascripts` folder by default as you proceed through the tutorial.

Deploy

When you are ready to deploy your application, run the script `script/build`. This will generate the host page, advanced compiled JavaScript and all of the resources that you will need to deploy. Output from `script/build` will go into the `out` directory at the root level of your project

Pasted from <https://github.com/brentonashworth/one/wiki/Workflow>

Project organization

Tuesday, February 14, 2012
8:52 AM

There are four top level directories in this project that a designer or developer will need to work with: `public`, `templates`, `src` and `test`.

public

```
public
├── 404.html
├── css
├── design.html
├── images
├── index.html
├── javascripts
└── js
```

The `public` directory contains files that are publicly available. With a running server, `404.html` can be reached at`http://localhost:8080/404.html`.

JavaScript source generated by the ClojureScript compiler will be put into `public/javascripts`.
The `public/js` directory contains static (i.e. not generated) JavaScript files; you can put third-party JavaScript files here if they will not be processed by the Google Closure compiler. Images and CSS should be put into `public/images` and `public/css`.

A designer may also want to change the links on the design page. This can be done by editing the `public/design.html` file and following the conventions in that file.

templates

```
templates
├── application.html
├── form.html
├── greeting.html
└── toolbar.html
```

The templates directory contains template HTML files which will become part of the running application. See the Design and templating page for more information about how to use these templates.

src

```
├── app
│   ├── clj
│   │   └── one
│   │       └── sample
│   │           └── ...
│   ├── cljs
│   │   └── one
│   │       └── sample
│   │           └── ...
│   └── cljs-macros
│       └── one
│           └── sample
│               └── ...
└── leiningen
└── lib
    ├── clj
```

```
│   └── one
│       └── ...
└── cljs
    └── one
        ├── browser
        │   └── ...
        └── ...
```

Source code is split into three directories: `app`, `lib`, and `leiningen`. The `app` directory is for code that is specific to the application that you are building. Initially, this directory contains code for the sample application. This code should be changed frequently. The `lib` directory contains library code that is not specific to the current application. The `leiningen` directory contains the code that implements the `lein bootstrap` and `lein git-deps` commands. See Dependencies for more information about how ClojureScript One integrates with Leiningen.

Even though the code in `lib` is "general purpose" and not specific to the application you are building, it may still need to be changed and should be easy for you to change. When useful changes are made to these libraries please consider contributing them back to ClojureScript One. Once this library code becomes mature, it will be extracted into an external library.

In both the `app` and `lib` directories, code is organized into `clj`, `cljs` and `cljs-macros` directories for Clojure, ClojureScript and macros.

test

Test code is located in the `test` directory. ClojureScript One provides support for testing ClojureScript code and Clojure code from your favorite Clojure testing framework. A few unit-level tests are provided as a starting point for your own testing needs.

Additionally, the project comes with an interesting integration test in the `test` directory under the `one.sample.test.integration` namespace. This test uses ClojureScript's Browser-connected REPL and a few utility functions to drive a browser session in much the same way that Selenium operates.

Design and templating

Tuesday, February 14, 2012
8:51 AM

Many Clojure web applications use Hiccup for HTML templating. If the programmer is also the designer, then Hiccup is ideal. However, most developers are bad at design. We need to work with people who are good at design and who don't need to care about Clojure. ClojureScript One proposes one approach to templating which allows designers to work with HTML, CSS and images without having to set an eye on Hiccup data structures or those pesky parentheses.

A typical designer workflow will look something like this:

1. Run `lein repl`
2. Type `(dev-server)` to launch the development server
3. Open a browser to http://localhost:8080 and click the **Design** button
4. Click on links to individual pages to assess the current look and feel of each page
5. Add/Edit files in the `templates` directory, add CSS and images to `public/css` and `public/images`
6. Update links to the above files by editing `public/design.html`
7. Goto step 4 and repeat until the designer is happy

Any changes made to template HTML files can be seen in the browser upon refresh.

Templates, partials, layouts, snippets etc.

ClojureScript One doesn't care how you create HTML. The templating system discussed here is not used by the running application. Its only purpose is to make designers' lives a little better.

To avoid repetition, two new HTML elements are introduced: `_include` and `_within`.

`_include` will include another HTML file in the current file and `_within` indicates that its content is to be included in another template. When using the `_within` tag, HTML ids are used to determine which elements are to be replaced.

A simple example will show how these new tags may be used.

Suppose we have a layout file `templates/layout.html` which contains the following HTML:

```
<html>
 <head>Example</head>
 <body>
  <_include file="menu.html"/>
  <div id="content"></div>
  <div id="footer"></div>
 </body>
</html>
```

Furthermore, suppose we have the following two files named `templates/menu.html` and `templates/example.html`.

```
<!-- templates/menu.html -->
<div>Navigation Menu</div>
<!-- templates/example.html -->
```

```
<_within file="layout.html">
 <div id="content">New Content</div>
 <div id="footer">A Footer</div>
</_within>
```

If we point the server at http://localhost:8080/design/example.html, the following file will be served:

```
<html>
 <head></head>
 <body>
  <div>Navigation Menu</div>
  <div id="content">New Content</div>
  <div id="footer">A Footer</div>
 </body>
</html>
```

Templating works with files in the `templates` directory and the `public` directory. To see other examples, look at the HTML files in these directories..

Including templates in the application

How does the HTML that the designers work with end up in the ClojureScript application? Enlive is an HTML transformation library for Clojure. We use it to extract snippets of HTML and insert them into the application.

You may be wondering how this is possible given that Enlive is a Clojure library and we are working in ClojureScript. That is a great question. The answer can be found in how ClojureScript macros work.

The snippets of HTML that are used in the application are created with the `snippets` macro shown below.

```
(defn snippet [file id]
 (render (html/select (html/html-resource file) id)))

(defmacro snippets []
 {:form (snippet "form.html" [:div#content])
  :greeting (snippet "greeting.html" [:div#content])})
```

Enlive is used to grab two chunks of HTML from the design HTML pages. In both cases it is a `div` element with a "content" id. Notice that`snippets` is a macro. In ClojureScript, macros are Clojure macros and run only at compile time. This means we can use any Clojure library from a macro.

The application host page

The JavaScript application we are creating must be hosted within an HTML page. The layout for the host page is contained in a file named`templates/application.html`. The Clojure namespace `one.host-page` contains the code for generating the host page for development mode, production mode and for building the deployment artifacts.

JavaScripts which are added to the host page can be configured in the `one.sample.config` namespace.

Pasted from <https://github.com/brentonashworth/one/wiki/Design-and-templating>

Development

Tuesday, February 14, 2012
8:52 AM

One of the main goals of this project is to provide the best possible development experience. Clojure developers expect to be able to modify code in a running application and the same should be true when working with ClojureScript.

Many people have complained about ClojureScript adding a compile step to JavaScript development. This is a valid complaint. In JavaScript, we are used to making changes and reloading the browser to see their effect. With ClojureScript, we can do even better.

ClojureScript One provides an environment where page reloads will reload ClojureScript code and Clojure code if any changes have been made. In addition to this, there is also a browser-connected REPL which allows you to make changes to a running application.

This project uses its own reloading scheme which is implemented in `src/lib/clj/one/reload.clj`. You may configure a list of Clojure files to watch for changes. If any file changes all of the watched files will be reloaded. If any ClojureScript file changes, or if any HTML template file changes, every ClojureScript file will be reloaded.

The specific Clojure files which are watched and reloaded can be configured in the `one.sample.config` namespace.

Reloading only the files that change is problematic. For example, if a macro from one file is used in another, and the macro implementation is changed, both files will need to be reloaded. If a protocol from one file is changed and another file implements this protocol, both files need to be reloaded. To keep things simple, this project will reload everything.

Reloading ClojureScript files when HTML templates change means that when you update an HTML file, the change is immediately visible in the ClojureScript application when the page is reloaded.

Most of the time, we don't need to resort to reloading the entire page because it is easy to add and update functions from the REPL. Reloading the page is most helpful when we are simultaneously making changes to the client and the server or we are modifying templates.

Browser-connected REPL

As briefly mentioned above, Clojure developers are used to making modifications to a running program. The REPL is not only used to experiment with existing code, but is also used to add new functions and modify existing functions. In a normal workflow, all development goes through the REPL.

To see just how interactive this can be, start the ClojureScript REPL as described on the Quick Start page and follow along with the examples below.

The example application provides a simple form which displays a greeting. As the state of the application changes, views which display the state are rendered. We can test this from the REPL:

```
(in-ns 'one.sample.model)
(swap! state assoc :state :greeting :name "James")
(swap! state assoc :state :init)
```

The `state` atom contains the state of the application. When the atom is updated, the views respond. These two state changes illustrate what happens in the view when valid data is entered and when the application is initialized.

The sample application uses an event dispatcher to fire and react to events. We can try adding new reactions and firing events from the REPL:

```
(in-ns 'one.sample.view)
(def alert-reaction (dispatch/react-to #{:greeting} (fn [_ d] (js/alert (str "Hello " (:name d))))))
(dispatch/fire :greeting {:name "Jim"})
(dispatch/fire :init)
```

With this new reaction in place, an alert will be displayed when the form is submitted.

Individual reactions can be removed. The call to `dispatch/react-to` returns the reaction which represents the association of an event to a reactor. The `dispatch/delete-reaction` function can be used to remove an individual reaction.

```
(dispatch/delete-reaction alert-reaction)
(dispatch/fire :greeting {:name "Jim"})
(dispatch/fire :init)
```

To remove all of the REPL code from the runtime environment, refresh the page.

Pasted from <https://github.com/brentonashworth/one/wiki/Development>

Testing

Tuesday, February 14, 2012
8:53 AM

The file src/lib/clj/one/test.clj contains the namespace one.test which implements support for testing ClojureScript code.

Clojure already has good testing frameworks, such as clojure.test, Midje and Lazytest. ClojureScript has a protocol for evaluation with implementations for the browser and Rhino. We can combine these two things to create a great testing environment for ClojureScript.

Let's experiment with this from the ground up. Start a Clojure REPL with lein repl and then start the development server.

```
lein repl
(dev-server)
```

Instead of connecting to the browser from a REPL, let's do something different. Let's create a browser evaluation environment which will allow us to evaluate arbitrary JavaScript code in the browser from Clojure.

```
(use '[cljs.repl :only (-setup -tear-down -evaluate)])
(use '[cljs.repl.browser :only (repl-env)])
(def eval-env (repl-env))
(-setup eval-env)
```

We have created the environment and will now need to connect to it. We could open a browser and navigate to the app but that's boring. Let's do it from the REPL.

```
(use '[clojure.java.browse :only (browse-url)])
(browse-url "http://localhost:8080/development")
```

We should now have an active connection. Let's test it by sending some JavaScript for evaluation.

```
(-evaluate eval-env "example.clj" 1 "1 + 1;")
;=> {:status :success, :value "2"}
```

The call to -evaluate takes the evaluation environment, a file name, line number and JavaScript string and returns a map containing the results of the evaluation.

What we would really like to be able to do is evaluate ClojureScript forms in this environment. We can use a function defined in ClojureScript, in the cljs.repl namespace, to help with this.

```
(use '[cljs.repl :only (evaluate-form)])
(evaluate-form eval-env {:context :statement :locals {}} "example.clj" '(+ 1 1))
;=> "2"
```

evaluate-form takes the evaluation (run time) environment, a map which represents the (compile time) environment in which the form will be evaluated, the file name and the form to evaluate. It returns a string representing the result of evaluation which can be read with the Clojure reader.

We are working at a very low level here, but hopefully you are starting to see where we are going with this. The one.test namespace builds on these capabilities to give us a more convenient way to do this.

```
(use '[one.test :only (evaluate-cljs cljs-eval *eval-env*)])
(evaluate-cljs eval-env '(+ 1 1))
;=> 2
```

`evaluate-cljs` is simpler to call and returns a Clojure value which we can use. We can also specify the namespace in which this form is to be evaluated.

```
(evaluate-cljs eval-env 'one.sample.model '@state)
;=> {:state :init}
```

In this example, we dereference the `state` atom in the `one.sample.model` namespace. The return value indicates that we are in the `:init` state.

The macro `cljs-eval` makes evaluating multiple forms in the same namespace a little easier, it will also ensure that the passed namespace has been loaded. Instead of passing in the evaluation environment, it relies on the value of the dynamic var *eval-env*. This allows the testing tool to determine which environment to run in.

```
(binding [*eval-env* eval-env]
  (cljs-eval one.sample.model (+ 1 1) @state))
;=> {:state :init}
```

As shown in the last example, multiple forms are passed to `cljs-eval`, only the return value for the last form will be returned.

Driving the browser

We now have the ability run arbitrary ClojureScript code in the browser from Clojure. Take a little break and reflect on how awesome that is. We'll wait.

This allows us to do some interesting things. If you have the form in view within the browser, try this.

```
(binding [*eval-env* eval-env]
  (cljs-eval one.sample.view
         (dispatch/fire [:editing-field "name-input"])
         (set-value! (by-id "name-input") "James")
         (dispatch/fire [:field-changed "name-input"] "James")))
```

This simulates the activity of a user filling in the form. First we fire the event that corresponds to the field gaining focus, then the value is entered and finally we fire the event that corresponds to the field losing focus, indicating that the value has been updated.

We might be tempted to use the `.focus` and `.blur` methods of the input field directly. However, in some browsers these events do not actually fire unless the browser window is the active window. Since we are presumably typing these commands into the REPL, it is the window containing our REPL that is active window, and not the browser. Here we are trying to make a more compelling demo, but this is something to be aware of when you are writing your automated tests.

Now let's click the button to submit the form.

```
(binding [*eval-env* eval-env]
  (cljs-eval one.sample.view
    (clojure.browser.dom/click-element :greet-button)))
```

To summarize, from Clojure we can evaluate arbitrary ClojureScript in the browser. This enables us to drive the browser, simulating user activity. Since we are running in the Clojure process which started the server, we can simultaneously simulate user interaction and test conditions on the client and server.

Example tests

ClojureScript One includes some example tests which demonstrate how to use this from the `clojure.test` testing framework.

Here is one example test:

```
(deftest test-enter-new-name
  (reset! *database* #{})
  (cljs-eval one.sample.view
          (dispatch/fire :init)
          (set-value! (by-id "name-input") "Ted")
          (fx/enable-button "greet-button")
          (clojure.browser.dom/click-element :greet-button))
  (cljs-wait-for #(= % :greeting) one.sample.model (:state @state))
  (is (= (cljs-eval one.sample.view (.innerHTML (first (nodes (by-class "name")))))
         "Ted"))
  (is (= (cljs-eval one.sample.view (.innerHTML (first (nodes (by-class "again")))))
         ""))
  (is (= (cljs-eval one.sample.model @state)
         {:state :greeting, :name "Ted", :exists false}))
  (is (true? (contains? @*database* "Ted"))))
```

This test will reset the `*database*` atom on the server, fill in and submit a form in the browser, wait until the `:state` key in the `state` atom on the client is `:greeting`, which indicates that the network activity has completed, and then test that the entered name appears on the visible web page, the `state` atom in the browser is correct and the `*database*` atom on the server contains the entered name.

See the file `test/one/sample/test/integration.clj` for an example of how to set up tests to open a browser and configure the evaluation environment.

Run `lein test` to run all the tests and see this in action

Pasted from <https://github.com/brentonashworth/one/wiki/Testing>

Event dispatching

Tuesday, February 14, 2012
8:53 AM

We sometimes write code like this:

```
(listen button
      "click"
      (process-form form-data))
```

When the button is clicked we process the form data. The problem with this code is that it is too specific. We are conflating the "who" and the "when" with the "what". All we need to do here is report what happened. We can decide how to respond to this later.

Using events, we might change the above code to look like this:

```
(listen button
      "click"
      (fire :greeting-form-submit form-data))
```

Here we are only reporting what happened. We are free to respond to this event in any number of ways without modifying this code. We can add logging, record events for later playback or update any part of the UI.

The namespace `one.dispatch` in the file `src/lib/cljs/one/dispatch.cljs` provides an implementation of a simple event dispatching system.

Events may be fired with an event-id and, optionally, data. Both the event-id and the passed data are arbitrary Clojure values. Reactor functions can be set up to react to specific events. Reactor functions are passed the event-id and the event data.

To see how this works, let's go through some examples interacting with the application from the REPL. To follow along, start a ClojureScript REPL using `lein repl` followed by `(go)` if you've not started one, and connect to the sample application. With an active REPL, move into the `one.dispatch` namespace.

```
(in-ns 'one.dispatch)
```

Creating reactions

Before we fire any events, we should set up a reaction. A reaction is created with the `react-to` function. The reaction below will show an alert displaying the message data sent in the event.

```
(def alert-reaction
  (react-to #{:test-event} (fn [event-id data] (js/alert data))))
```

`react-to` takes an event predicate function and a reactor function and returns a reaction. When an event is fired, the event-id will be passed to the event predicate function. If this function returns `true`, then the reactor function will be called passing the `event-id` and `data`.

In the example above, the predicate function is a set. Sets are functions which take a value and return that value if it is in the set and `nil` if it is not in the set.

We can test this by firing an event, as shown below.

```
(fire :test-event "foo")
```

The `fire` function takes an `event-id` and, optionally, a value. Both the `event-id` and the `data` can be any ClojureScript value.

The reaction that is returned from the call to `react-to` may be used to delete the reaction.

```
(delete-reaction alert-reaction)
```

If we now try to fire the same event that was fired above, we will notice that nothing happens.

Sometimes we would like to react to something a specific number of times. For example, we may have some resource that we want to clean up whenever the state of the application changes. The three argument version of `react-to` takes a number as its first parameter. This is the number of times that the reaction is run before being deleted.

```
(react-to 1 #{:one-time} (fn [event-id data] (js/alert data)))
(fire :one-time "foo")
(fire :one-time "bar")
```

In the above example, the reaction is set up to run once. The first fired event will be displayed in an alert box and the second will not.

Events in the sample application

Even though the sample application is small, there are a lot of interesting uses of events.

The input form provides instant feedback for every state of the form and for every state transition. For example, when data is entered in a field, the value will be validated and the form will display an error message. Events are used to keep the validation code out of the view.

Views are rendered when application state changes. Events are used to report that application state has changed.

We can view all fired events by enabling logging. To do this, evaluate the following form:

```
(one.logging/start-display (one.logging/console-output))
```

This will cause all events to be logged to the JavaScript console. In the browser, open a console and interact with the application. Each fired event will be printed to the console.

Reactions may be added anywhere in the application, allowing new features to be added without modifying existing code.

Both history and logging are implemented in this way.
See `src/app/cljs/one/sample/logging.cljs` and `src/app/cljs/one/sample/history.cljs` for more information.

Logging

Tuesday, February 14, 2012
8:55 AM

The Google Closure library provides extensive support for logging. A simple Clojure wrapper for this API is provided in the file `src/lib/cljs/one/logging.cljs` which contains the `one.logging` namespace.

The Library

Before explaining the details, let's start with a simple example. Start the sample application and a ClojureScript REPL. Open a JavaScript console so that you will be able to see log messages as they are printed below and then evaluate the following forms.

```
(in-ns 'one.logging)
(def example-logger (get-logger "wiki.examples"))
(info example-logger "Hello")
```

In the example above, we create a logger, giving it a name, and then use the logger to log the message "Hello" at the `info` logging level. You may have noticed that nothing happened. That is actually a feature. Log messages are not displayed until you explicitly indicate where they should be displayed and at what level.

Currently, log messages can be displayed in the JavaScript console or in a separate window.

Use the `console-output` function to create a log viewer which will display log messages in the JavaScript console. The `start-display` function will cause messages to start printing in the console.

```
(def console-out (console-output))
(start-display console-out)
(info example-logger "Hello")
```

You should now see log messages being printed to the console. The default format for printing log messages is:

```
[time] [logger name] message
```

The time that is displayed in the number of seconds since the application started.

Log Clojure data

You will often want to log Clojure data to the console.

```
(info example-logger {:a "message"})
```

This prints something like...

```
[138.626s] [wiki.examples] [object Object]
```

...which is not very helpful. To achieve the desired result, use `pr-str` to print the Clojure data to a string before printing to the console.

```
(info example-logger (pr-str {:a "message"}))
```

One of the many joys that you will experience while working with ClojureScript is the joy of copying a large printed data structure from the console into the REPL and then working with it as you would with any other Clojure data.

You can tell a log viewer to stop printing by calling the `stop-display` function.

```
(stop-display console-out)
```

If you executed the above command, you can reverse it again simply by running `(start-display console-out)` as before.

logging levels

In the examples above, we have been logging at the `info` level. The logging library supports seven levels:

- severe
- warning
- info
- config
- fine
- finer
- finest

By default, the output logging level is set to `config`. The current logging level will print all log messages at that level or higher.

The following form will print all log messages except for "FINE".

```
(start-display console-out)
(do (severe example-logger "SEVERE")
  (warning example-logger "WARNING")
  (info example-logger "INFO")
  (config example-logger "CONFIG")
  (fine example-logger "FINE"))
```

Use the `set-level` function to change the logging level for a logger.

```
(set-level example-logger :fine)
```

Try logging all of the messages above with different logging levels.

Fancy logger window

If the browser you are using does not have a console window or your would like to view log messages in a separate window, you may use the fancy logging view.

```
(def fancy-out (fancy-output "Example Output Window"))
(start-display fancy-out)
```

Use the `start-display` and `stop-display` functions to start and stop printing to this window.

The sample application

The sample application is set up to log all events. This is implemented in the file `src/app/cljs/one/sample/logging` which contains the `one.sample.logging` namespace.

Logging of all events is added to the application without having to modify any existing code.

```
(dispatch/react-to (constantly true)
                   (fn [t d] (log/info logger (str (pr-str t) " - " (pr-str d)))))
```

In the code above, a reaction is created which will react to any event and log it at the `info` logging level.

By default, log messages are not printed anywhere. To start viewing events in the console, create a console log viewer and start it as described above.

```
(in-ns 'one.sample.logging)
(log/start-display (log/console-output))
```
You should now see events as you interact with the application.

Pasted from <https://github.com/brentonashworth/one/wiki/Logging>

History

Tuesday, February 14, 2012
8:55 AM

The file src/lib/cljs/one/browser/history.cljs contains the one.browser.history namespace which implements basic support for working with the browser's history.

Traditional web applications allow users to navigate between pages by clicking on hyperlinks, using the browser's navigation buttons and entering URLs in the address bar. Single-page JavaScript applications do not have pages, and yet browsers still allow all of these forms of navigation. Providing support for this type of navigation can help make an application feel more familiar.

Google Closure supports working with history and responding to navigation events generated by the browser. The one.browser.history namespace provides a very simple wrapper around this functionality.

The sample application has two main views: the form where a user may enter their name and a greeting view where a greeting message is displayed. In the sample application, when the form is displayed, the #form document location fragment is added to the end of the URL.

http://localhost:8080/development#form

When the greeting page is displayed the #greeting document page fragment is added to the end of the URL.

http://localhost:8080/development#greeting

History changes are reflected in this document location fragment. As changes are made to the history, these fragments (also known as tokens) are stored internally in a stack.

Users may navigate this stack by clicking the "Back" or "Forward" button or by entering a URL in the address bar which contains a document location fragment.

History in ClojureScript

To work with history from ClojureScript, you must first create a history object. This must be done before the host page has finished loading. The history object is initialized with a function that will be called every time the history is changed.

```
(use '[one.browser.history :only (history set-token)])
(use '[one.sample.history :only (nav-handler)])
(history nav-handler)
```

nav-handler is a function defined in the one.sample.history namespace of the sample application. It fires the token it receives when a navigation event occurs. Valid tokens for this application are :form and :greeting which correspond to events which move the application to the :form or :greeting state.

```
(defn nav-handler [{:keys [token navigation?]}]
  (when navigation?
    (dispatch/fire token)))
```

The history may be modified from ClojureScript with the set-token function.

```
(set-token :something)
```

When the user navigates by clicking the "Back" button, or entering a URL with a document location fragment, the nav-handler function will be called, passing a map with :token, :type and :navigation? keys. An application is free to respond to these events in any way.

Pasted from <https://github.com/brentonashworth/one/wiki/History>

Remoting

Tuesday, February 14, 2012
8:55 AM

The file src/lib/cljs/one/browser/remote.cljs contains the one.browser.remote namespace which provides a single function, request, for making HTTP requests to remote servers.

At a minimum, the request function takes id and url arguments. The id must be unique for active requests. That is, you may reuse a request's ID, but only once the request has ended.
The url parameter is the URL to connect to. By default, the GET method will be used but the method can be specified by using the :method keyword argument.

The request function simply wraps the send function form Google Closure's goog.net.XhrManager object which has the following description:

"Registers the given request to be sent. Throws an error if a request already exists with the given ID. NOTE: It is not sent immediately. It is queued and will be sent when an goog.net.XhrIo object becomes available, taking into account the request's priority."

Several Valid calls to request are shown below.

```
(request :my-id "http://localhost/api?a=1")
(request :my-id "http://localhost/api"
      :method "POST"
      :content "a=1")
(request :my-id "http://localhost/api"
      :method "POST"
      :content "a=1"
      :on-success #(js/alert (pr-str %)))
```

If you have a running ClojureScript REPL connected to the sample application, you can try this out:

```
(in-ns 'one.browser.remote)
(request :add-name "http://localhost:8080/remote"
      :method "POST"
      :on-success #(js/alert (:body %))
      :content (str "data=" (pr-str {:fn :add-name :args "James"})))
```

The first time you evaluate this function, you should see the map {:exists false} in the alert box, after that you will see {:exists true}. The add-name service on the server will add a name to a set and return {:exists false} if the name was not already in the set and{:exists true} if it was.

Other optional parameters

Other than the parameters described above, you may also use :headers, :priority and :retries as keyword parameters.

:headers is a map of HTTP request headers to use for this request. For example:

```
{"Content-Type" "text/html"}
```

:priority is the priority of this request.

:retries is the maximum number of times the request should be retried.

Callbacks

Two callbacks can be provided which will allow you to process the results of a request

asynchronously: `:on-success` and `:on-error`.

The `:on-success` callback will be called if the response code is 200 and the `:on-error` callback will be called otherwise. This behavior is not technically correct - for example, 201 (Created) is a success code, but will trigger the `:on-error` callback. This has been fixed in Google Closure, but ClojureScript is currently using a version of Google Closure that does not include the fix. ClojureScript will be updated in the future to include this fix.

The data which is passed to the callback functions contains the keys `:id`, `:body`, `:status` and `:event`. `:id` is the ID that was passed to the `request` function. `:body` is the result that was sent back from the server, `:status` is the HTTP response status and `:event` is the JavaScript event object which contains more information about this response. From the event object you can get the goog.net.XhrIo object and then call any of the functions defined on it.

The example below shows how to get the "Content-Type" header from the response.

```
(defn show-content-type [{e :event}]
  (js/alert (.getResponseHeader (.xhrIo e) "Content-Type")))
(request :add-name "http://localhost:8080/remote"
       :method "POST"
       :on-success show-content-type
       :content (str "data=" (pr-str {:fn :add-name :args "James"})))
```

See the documentation for goog.net.XhrManager for more information.

Other examples

See the Controller documentation for an example of how this is used in the sample application.

Pasted from <https://github.com/brentonashworth/one/wiki/Remoting>

Animation

Tuesday, February 14, 2012
8:55 AM

The file src/lib/cljs/one/browser/animation.cljs contains the one.browser.animation namespace which implements basic effects and animations.

An animation is a composition of one or more effects or animations.

Effects as data

A simple effect can be described with five pieces of information: the effect name, the start value, the end value, time and acceleration. The values of start and end will depend on the kind of effect. Effects can be described with a simple Clojure map. For example: if we want to create an effect that moves the label in the text field of the sample application, we could create a slide effect.

```
{:effect :slide, :end [10 10], :time 500}
```

This describes an effect which will slide an element to the position [10 10] in 500 milliseconds. Notice that we do not need to provide all of the values. Some missing values have defaults and some will be calculated. In this example the start position is assumed to be the current position of the element that will be moved.

In some situations, we only know that we want to move something relative to its current location.

```
{:effect :slide, :up 40, :time 200}
```

The slide effect allows us to specify, :up, :down, :left and :right amounts and the positions will be calculated for us.

Binding and running effects

As we move through the rest of this page, you will want to follow along. Start the sample application and open a ClojureScript REPL. Evaluate the following form to enter the one.browser.animation namespace.

```
(in-ns 'one.browser.animation)
```

Effects will visually change an HTML element and so must be bound to an element before they can be run. In the terminology of this library, once an effect is bound, it becomes an animation. The bind function is used to associate effects with elements and produce animations.

In the example below, we use Domina to find the label element that we would like to animate. We then bind the effect to the element and call the start function to start the animation.

```
(def label (get-element "//label[@id='name-input-label']/span"))
(start (bind label {:effect :slide, :up 40, :time 200}))
```

Running this animation will move the label above the text field. We can move the label back into the field by moving it the same distance in the opposite direction.

```
(start (bind "//label[@id='name-input-label']/span"
        {:effect :slide, :down 40, :time 200}))
```

Notice that in this example, the xpath string is passed directly to bind. bind can be passed an element, a keyword which represents the CSS ID of an element or an xpath string.

Composing effects

When you click in the text field, you will notice that two things happen at the same time: the label moves up and the color changes. This is an example of two effects being applied in parallel. The two effects are :slide and :color.

```
{:effect :color, :end "#53607b", :time 200}
{:effect :slide, :up 40, :time 200}
```

The bind function can take any number of effects and will run them in order. In the examples below, the label's color will change and then the label will be moved. Each change takes 1 second (the default) so that you can clearly see what is going on.

```
(start (bind label {:effect :color, :end "#53607b"}
               {:effect :slide, :up 40}
               {:effect :color, :end "#BBC4D7"}
               {:effect :slide, :down 40}))
```

In the sample application, the color and slide effects happen at the same time. To run effects in parallel, we put them in a vector.

```
(start (bind label [{:effect :color, :end "#53607b"}
                    {:effect :slide, :up 40}]
                   [{:effect :color, :end "#BBC4D7"}
                    {:effect :slide, :down 40}]))
```

The bind function accepts any number of vectors and maps and will run them sequentially. A map represents a single effect and a vector represents multiple effects to be run in parallel. In the first example of bind above, we passed four maps which were run sequentially. In the second example, we passed two vectors, each containing two maps. The two vectors are effects which were run sequentially. The maps in each vector represent effects which were run in parallel.

Composing animations

Using bind to create complex animations is simple but it only allows us to create animations for a single element. Sometimes we may want to create animations which involve multiple elements.

Imagine that we would like to create an animation that moves the field label up and down while changing its color. In this example we get the label element and record is initial color. We then create an animation for the label which moves it up while changing its color to red, and then moves it down while changing its color back to its original color.

```
(def label (get-element "//label[@id='name-input-label']/span"))
(def label-color (color label))
(def red [255 0 0])
(def move-label (bind label
                   [{:effect :slide, :up 200, :time 2000, :accel :ease-out}
                    {:effect :color, :end red, :time 2000}]
                   [{:effect :slide, :down 200, :time 2000, :accel :ease-in}
                    {:effect :color, :end label-color, :time 2000}]))
(start move-label)
```

Next, imagine that we would like to change the background color of the text field. We first get the input element with the ID :name-input. We then define some colors and record the original background color of the field. The animation will change the background color to red, green, blue and then back to the original color.

```
(def input (get-element :name-input))
(def green [0 255 0])
(def blue [0 0 255])
```

```
(def input-bg-color (bg-color input))
(def background (bind input
                      {:effect :bg-color, :end red}
                      {:effect :bg-color, :end green}
                      {:effect :bg-color, :end blue}
                      {:effect :bg-color, :end input-bg-color}))
(start background)
```

Finally, we would like to make the button change its size several times. We get the button and record its size. We then create an animation that makes several changes to the button's size. All of the other animations last about 4 seconds and we would like this one to take the same amount of time. We map the effects over a function that sets the time for each effect to 4000/6.

```
(def button (get-element :greet-button))
(def button-size (size button))
(def button-dance (apply bind button
                    (map #(assoc % :time (quot 4000 6))
                       [{:effect :resize, :end [150 150]}
                        {:effect :resize-height, :end 200}
                        {:effect :resize-width, :end 200}
                        {:effect :resize-height, :end 150}
                        {:effect :resize-width, :end 150}
                        {:effect :resize, :end button-size}])))
(start button-dance)
```

The functions `serial` and `parallel` may be used to take many animations and compose them into a single animation. `serial` will compose them into a single animation which will run each animation in sequence. `parallel` will compose them into a single animation which will run each animation in parallel.

```
(start (serial move-label background button-dance))
(start (parallel move-label background button-dance))
```

Animation events

Events can be used with animations to trigger actions when an animation begins or finishes. In the example below we define a var with the complex parallel animation from above. We then listen once for a `finish` event and show an alert box.

```
(def a (parallel move-label background button-dance))
(event/listen-once a
                   "finish"
                   #(js/alert "Animation Finished."))
(start a)
```

You may listen for `begin` and `finish` events.

The effects

Available effects include: `:color`, `:fade`, `:fade-in`, `:fade-out`, `:fade-in-and-show`, `:fade-out-and-hide`, `:slide`, `:swipe`,`:bg-color`, `:resize`, `:resize-width` and `:resize-height`.

Many of these effects are demonstrated in this page. For more detail and examples, please refer the source file`src/lib/cljs/one/browser/animation.cljs`.

Acceleration

Each effect can have an associated acceleration function. An acceleration function is a function which takes a number in the range [0 1] and returns a number in the same range. There are three built-in acceleration functions: `:ease-in`, `:ease-out` and `:ease-in-and-out`. These functions can be used in

effects by associating the `:accel` key with one of these keywords.

```
{:effect :slide, :up 200, :time 2000, :accel :ease-out}
```

You may also implement your own acceleration functions.

Pasted from <https://github.com/brentonashworth/one/wiki/Animation>

The Sample Application

Tuesday, February 14, 2012
8:55 AM

The sample application should provide enough information to help you get started building your own applications. It is small enough to comprehend in a short time and large enough to demonstrate how to deal with many issues that will come up in a much larger application.

The application presents a form which takes a name and then displays a greeting message. While performing this simple function, it demonstrates several interesting things about building this kind of application in ClojureScript. Some of these things include:

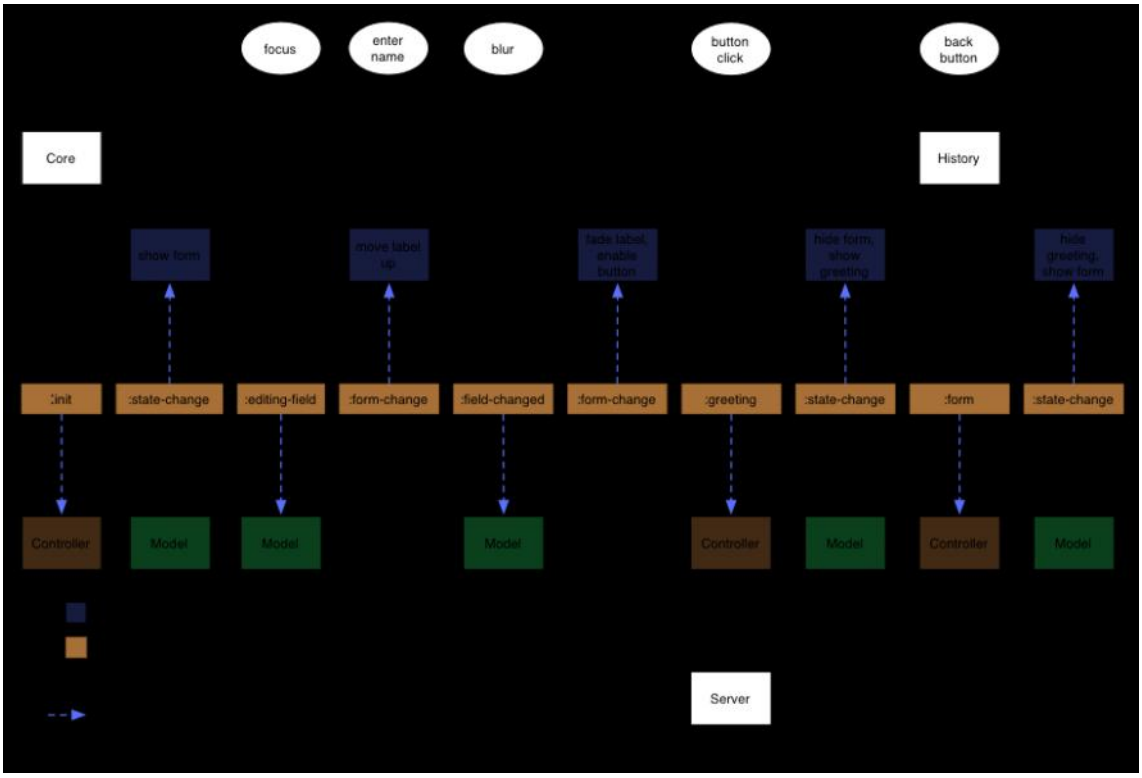
- Code organization
- Using events to decouple components
- The importance of plain Clojure data
- Sending Clojure data to the server and back
- Creating highly interactive user interfaces
- Keeping application code out of the view

Overview

Although there are things named Model, View and Controller in this application it is not exactly MVC as commonly practiced. The paradigm used in this application is a mix of MVC, Functional reactive programming and Dataflow programming.

The backbone of the application is an event dispatching system. Normally, when something happens, we call a function to do something about it. For some key events in the system, it is better to simply report what has happened and not make a decision about how this event should be dealt with. That decision can be made later, without changing the code which reports the event. In a multi-threaded environment we would use a queue for this. In a single-threaded environment we are essentially using a queue which only holds a single element. The event dispatching system implements the ability to report events and to provide functions which will react to events.

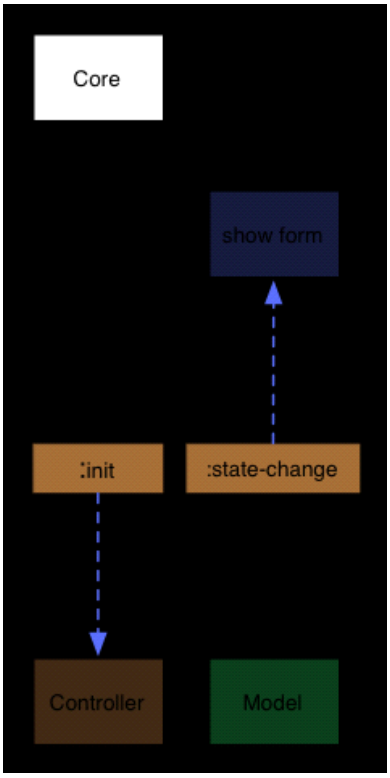
There are many advantages to this kind of decoupling but the one disadvantage is that it can be a bit harder to follow the flow of execution (although doing this is probably the wrong way to think about this type of program). The diagram below shows a typical interaction with the sample application and tracks the flow of execution.



As you can see, the orange squares down the middle of the diagram show the stream of events that are generated as the application runs. Each event contains data that completely describes that event. Instead of hiding this data away in function calls, we expose it. This stream of events is a collection of Clojure data which represents what has happened as the application runs. This data can be the basis of many interesting features. See the one.sample.logging and one.sample.history namespaces for examples of how useful features can be added without modifying any existing code. If we were to record these events (which could be done without modifying existing code) they could be: queried to find bugs, used to replay user actions to reproduce problems or used to test specific components independent of others.

A quick walk-through

To get a better feel for the sample application, let's walk through what happens from the point where the application starts, to the rendering of the initial view.



The first thing that happens when the application starts is the firing of an `:init` event. This function is located in the `one.sample.core` namespace.

```
(defn ^:export start
  []
  (dispatch/fire :init))
```

The controller (`one.sample.controller`) responds to `:init`, `:form` and `:greeting` events by calling the corresponding `action`multimethod.

```
(dispatch/react-to #{:init :form :greeting}
                   (fn [t d] (action (assoc d :type t))))
```

When an `:init` event is fired, the `state` atom will be reset to contain `{:state :init}`.

```
(defmethod action :init [_]
  (reset! state {:state :init}))
```

The `state` atom is being watched, and when it changes, the watcher function will fire a `:state-change` event.

```
(def state (atom nil))

(add-watch state :state-change-key
           (fn [k r o n]
             (dispatch/fire :state-change n)))
```

Views react to `:state-change` events by rendering a view which displays the current state of the application.

```
(dispatch/react-to #{:state-change} (fn [_ m] (render m)))
```

When a `:state-change` event is fired, and the current state is `:init`, the `render` function will initialize the

user interface.

```
(defmethod render :init [_]
  (fx/initialize-views (:form snippets) (:greeting snippets))
  (add-input-event-listeners "name-input")
  (fx/disable-button "greet-button")
  (event/listen (by-id "greet-button")
          "click"
          #(dispatch/fire :greeting
                  {:name (value (by-id "name-input"))})))
```

This view will add the snippets for the form and greeting page to the DOM, configure event listeners for the name input field and disable the submit button. Finally, an event listener is added to the submit button which will fire a `:greeting` event which contains the name entered in the input field.

The `fx` alias above refers to the `one.sample.animation` namespace where all of the animations are defined.

For more details about the sample application, see Model, View and Controller.

Pasted from <https://github.com/brentonashworth/one/wiki/The-sample-application>

Model

Tuesday, February 14, 2012
8:58 AM

The file src/app/cljs/one/sample/model.cljs contains the one.sample.model namespace that implements the model for the sample application.

The idea behind models in this application is that they represent the current state of the application. When a model is updated, views may need re-rendering to display new information to the user.

To avoid coupling the view directly to the models, we fire events when a model's state changes.

ClojureScript provides direct support for this in atoms and watchers. In the one.sample.model namespace there are two atoms: state and greeting-form. The state atom represents the current state of the entire application. It contains a key, :state, which can have a value of :init, :form or greeting.

When the state atom is updated, a :state-change event is fired by its watcher. Functions in the one.sample.view namespace will react by rendering either the form or greeting view. The :init state will cause everything to be re-rendered.

We can experiment with this from the REPL. Start a ClojureScript REPL and open the sample application in the browser. Once you have an active REPL, enter the one.sample.model namespace. You may also want to log all events to the console.

```
(in-ns 'one.sample.model)
(one.logging/start-display (one.logging/console-output))
```

If we update the state atom, setting the state to :greeting and the name to "James", the greeting view will be displayed. Changing the state back to :init will re-display the form.

```
(swap! state assoc :state :greeting :name "James")
(swap! state assoc :state :form)
```

If you are following along, you may have noticed that the field label is moving down the page. That is because we are not following a valid sequence of events. In real usage, the label would always be above the field when the form is displayed. Moving the label down will put it into the correct position. We can fix this by setting the state to :init which will start the application from the building.

```
(swap! state assoc :state :init)
```

The greeting-form atom represents the state of the form that accepts the user's name. Both the state of the entire form and the state of the fields are represented. One possible state of the form is represented by the map below.

```
{:status :editing
 :fields {"name-input" {:status :error
                        :value "a"
                        :error "Name is too short!"}}}
```

This state means that the form is currently being edited and the name-input field has an error. There is a specific view of this form that corresponds to this state.

A ClojureScript watcher will fire a :form-change event when this atom is changed. The data associated with the event will contain both the old and new state of the atom. The reactor function for each field can use the old and new state to calculate the state transition which has just occurred and perform the appropriate animation.

In the browser, click in the text field, type the letter "a" and then click outside of the field. We can now

inspect the value of the `greeting-form` atom.

```
@greeting-form
```

Notice that the value that is printed is the same as the map shown above but with a longer error message. What do you think would happen if we directly update the `greeting-form` atom?

```
(swap! greeting-form assoc :status :finished)
```

When we do this, the "Done!" button is enabled as if the form has been completed. The atom was updated and a `:form-change` event was fired. The view is trying to render the inconsistent state of the form. We can fix this by changing `:status` back to `:editing`.

```
(swap! greeting-form assoc :status :editing)
```

If the state becomes inconsistent, the view will reflect that. Apart from defining atoms and adding watches, the rest of the code in the `one.sample.model` namespace is concerned with making sure the state is always consistent.

When the field gains focus, the event `[:editing-field "name-input"]` is fired. When the field loses focus, the event-id `[:field-finished "name-input"]` is fired. When the field is changed, the event-id `[:field-changed "input-field"]` is fired. The model reacts to each of these events. The reaction for the first event is created in the code shown below.

```
(dispatch/react-to (fn [e] (= (first e) :editing-field))
                   (fn [[_ id] _] (set-editing id)))
```

If the first element of the event-id for a fired event is `:editing-field`, call a reactor function which will update both the status of the field and the form to `:editing`.

When the first element of the event-id is `:field-changed`, the field will be validated before the `greeting-form` state is changed. If the field is not valid, that will be represented in the state and then rendered in the view.

Notice that the view doesn't need to know anything about where and when validation occurs. It only needs to know how to render each state.

Pasted from <https://github.com/brentonashworth/one/wiki/Model>

View

Tuesday, February 14, 2012
8:58 AM

The file src/app/cljs/one/sample/view.cljs contains the one.sample.view namespace which is responsible for rendering views in the sample application.

A "view" is anything that changes the user interface. This could mean manipulating the DOM, changing a CSS class on an element or running an animation.

View rendering actions occur in a reactive way based on changing states over time in the model. Views are only changed when the application state changes. When this happens, Clojure provides both the old and new value. This allows us to easily render different views based on both the current and previous state.

All views react to either a :state-change or :form-change event. :state-change events are fired when the state of the application changes.:form-change events are fired when the state of the form changes.

There are three functions in one.sample.view which are responsible for rendering views: render, render-form-field and render-button. Each of these is implemented as a multimethod that dispatches based on some function of the current state.

The easiest of the three functions to understand is render because it dispatches on the value associated with the :state key in the application state. To follow along, start a ClojureScript REPL and enter the one.sample.view namespace.

```
(in-ns 'one.sample.view)
```

Rendering form and greeting screens

When the application starts, :state is set to :init. A reaction is defined that "reacts" to a :state-change event by calling the renderfunction, like so:

```
(dispatch/react-to #{:state-change} (fn [_ m] (render m)))
```

```
;=> {:max-count nil, :event-pred #{:state-change}, :reactor #<function ...>}
```

From the REPL, we can see what happens when the render function is called with this state.

```
(render {:state :init})
```

This view will remove everything from the DOM and then re-add the elements required by the application.

Rendering the form

The form for this application is an example of how dynamic a user interface in a JavaScript application can be. The idea behind this form is that it is validated while it is being edited and that the form cannot be submitted until it contains valid data.

The function add-input-event-listeners, adds listeners to the input field that will fire events to indicate that the field is being edited or that the field's value has changed. The model is listening for these events and will validate the input and update the form state accordingly.

The view reacts to :form-change events which are fired when the form's state changes. When a :form-change is received, the data associated with the event contains both the old and new state of the form. This data is transformed before calling the render-form-field andrender-button functions.

The render-form-field function receives a map with :transition and :id keys and will run the correct animation depending on the state transition which has just occurred.

```
(render-form-field {:transition [:editing :empty] :id "name-input"})
;=> true
```

The transition above means that the field is no longer being edited and is empty..

The render-button function receives a vector of two elements which represents the state transition of the entire form.

```
(render-button [:editing :finished])
```

The Vector [:editing :finished] means that the form has transitioned from being edited to being finished and can be submitted.

Go ahead and try calling the render function with other valid states.

```
(render {:state :greeting :name "James"})
(render {:state :form})
(render {:state :init})
```

Where does all the HTML for the view come from?

The first definition in the one.sample.view namespace looks like this:

```
(def snippets (snippets/snippets))
```

We can inspect the contents of snippets from the REPL.

```
snippets
;=> {:form "<div id=....>" :greeting "<div id=...>"}
```

It contains a map of keywords to strings of HTML. Were do these strings come from?

snippets contains the value returned from calling (snippets/snippets). This is a call to the snippets macro in the one.sample.snippetsnamespace.

```
(defn- snippet [file id]
  (render (html/select (html/html-resource file) id)))

(defmacro snippets
 "Expands to a map of HTML snippets which are extracted from the
 design templates."
 []
 {:form (snippet "form.html" [:div#form])
  :greeting (snippet "greeting.html" [:div#greeting])})
```

The macro uses Enlive to extract the required HTML from the HTML templates in the templates directory and return them in a map. Enlive does not work in ClojureScript but does work from Clojure. Macros in ClojureScript are just regular Clojure macros.

Each time the ClojureScript application is compiled, the templates defined in the snippets function will be pulled into the ClojureScript application. This scenario allows the separation of the development activities from the design activities.

Because of the way reloading is implemented, if you edit a template and then refresh the page, the changes to the template will be visible immediately.

Pasted from <https://github.com/brentonashworth/one/wiki/View>

Controller

Tuesday, February 14, 2012
8:58 AM

The file src/app/cljs/one/sample/controller.cljs contains the one.sample.controller namespace which implements the controller for the sample application.

Models and Views are easy to understand but what do we mean here by Controller? Events generated by the application may result in changes to the internal application state or to remote services. Whenever coordination needs to be done to update multiple models or make requests to external services, those changes are handled by the controller.

In the sample application, the state of the application can change between :init, :form and :greeting. The :init and :form state only require a simple change to the state atom. The :greeting state requires a network round-trip to send the name to the server and find out if that name has already been entered.

Controllers are all about coordination of data flowing through the system. Models and Views are end-points where something definitive happens: Models are changed, Views are rendered. Controllers are always passing data along to something else.

Talking to a Clojure server

The sample application sends data to a backend service written in Clojure. This is done in a function named remote.

We can try this function from the ClojureScript REPL.

```
(in-ns 'one.sample.controller)
(remote :add-name "James" #(js/alert (pr-str %)))
```

remote accepts a keyword which represents the dispatch value for the function on the server, data (a name) and a callback function. In the example above the callback will show an alert which displays the printed value returned from the server.

The remote function is implemented on top of request from one.browser.remote.

```
(defn remote [f data on-success]
  (request f (str (host) "/remote")
           :method "POST"
           :on-success #(on-success (reader/read-string (:body %)))
           :on-error #(swap! state assoc :error "Error communicating with server.")
           :content (str "data=" (pr-str {:fn f :args data}))))
```

The server side is implemented by defining a route with Compojure.

```
(defroutes remote-routes
  (POST "/remote" {{data "data"} :params}
        (pr-str
         (remote
          (binding [*read-eval* false]
            (read-string data))))))
```

Please note that remote in the second example has nothing to do with the remote function in the first example. One is a function on the client and the other a function on the server. They do not need to have the same name.

The main thing to notice here is that we are using pr-str and read-string on both the client and server to

serialize and deserialize Clojure data for transport over the network.

In the browser, the string that constitutes the content of the `POST` request is created in the following way:

```
(str "data=" (pr-str {:fn f :args data}))
```

An example of some actual content is shown below.

```
(str "data=" (pr-str {:fn :add-name :args {:name "James"}}))
;=> data={:fn :add-name, :args {:name "James"}}
```

On the server the value of `data` is read with `read-string` and passed to the `remote` function which will dispatch to the correct method based on the value associated with the `:fn` key. The return value from the call to `remote` is then printed with `pr-str`.

If the connection is successful, the following function is called in the browser.

```
#(on-success (reader/read-string (:body %)))
```

This calls the user-provided `on-success` function, passing it the result of reading the body of the response with `read-string`.

From this simple example, I hope it is obvious that being able to both print and read Clojure data from Clojure and ClojureScript greatly simplifies client-server applications. Data is central in Clojure. Its rich data literals allow us to represent complex data structures including Clojure source code. Being able to send this data over a network without having to encode it in another format eliminates much of the hard work associated with client-server programming.

Pasted from <https://github.com/brentonashworth/one/wiki/Controller>

The Server

Tuesday, February 14, 2012
8:59 AM

When creating an application with ClojureScript One, the end goal is to produce a single JavaScript file containing our program. This program may talk to a backend service which will require a server. During development, we also need a server but for very different reasons. We need a server that helps us create our application.

ClojureScript One comes with two servers. One for development and another much smaller server which can be used to serve the service for the sample application in production.

As you work with ClojureScript One, you should feel free to modify these servers in any way. They are meant to be a helpful starting point.

The development server is implemented in the file `src/app/clj/one/sample/dev_server.clj` which contains the `one.sample.dev-server` namespace.

In serving the development environment, this server provides many helpful functions:

- Three views of the application
- The design view
- A host page for the development view
- A host page for the production view
- ClojureScript One menu
- Templating support
- Proper encoding of JavaScript files
- Code reloading for both Clojure and ClojureScript

The application which is served by the development server is a tool. It is a tool which simultaneously hosts the application that you are building and helps you build it. One of the powerful things about this tool is that it can be customized using the same skills which you use to build your application.

The first thing that you see when going to the application is the `index.html` page which simply confirms that you have started the application and gives you an overview of the three views of the application. This page should be customized to meet your own needs.

The three views of the application are Design, Development and Production. Every view uses the same HTML template which is located in the file `templates/application.html`.

Design view

The Design view allows developers and designers to view static HTML resources outside of the context of the application. To support this view the server uses the templating system built into ClojureScript One and implemented in the file `src/lib/clj/one/templates.clj`. For more information about how templates work, see the Design and templating page.

The server supports templating for HTML files in both the `templates` directory and the `public` directory.

The Design area allows you to view template files as static HTML. The main design page is named `design.html` and is located in the `public` directory. Any other route which begins with `/design` is

mapped to files in the `templates` directory.

One convenient feature provided by the development server is that all HTML files can reference resources in the `public` directory in the same way no matter where the HTML files are located. For example, in both the `public` and `templates` directory, HTML files can link to the CSS file at `public/css/one.css` using:

```
<link href="css/one.css" rel="stylesheet">
```

Development view

This is a tool for creating a ClojureScript application. That application needs to live in a host HTML page. In most projects you need to have two host pages: one for the development version of the application and the other for the production version. In ClojureScript One, these pages are generated automatically.

Support for generating host pages is implemented in the file `src/lib/clj/one/host_page.clj`.

The development view uses the file `templates/application.html` as the host page, appending JavaScript files to the body of this page. This view must include script tags for the main JavaScript file for the application and the Google `base.js` file. It must also explicitly require each namespace that will be used by the application (Google Closure will calculate dependencies, so only top-level namespaces need to be required).

After all of the require statements, calls must be made to the JavaScript functions which will start the application. In the development view, the following calls are made:

```
one.sample.core.start();
one.sample.core.repl();
```

The call to `start` will start the application and the call to `repl` will initiate the client side of the connection to the browser-connected REPL.

Production view

The production view also uses the file `templates/application.html` as the host page but only needs to add the advanced compiled application JavaScript and the call to `start`.

Templating

As mentioned above, templating is applied to all HTML files served from the `public` or `templates` directory. Any file which is referred to in a template in a `_within` or `_include` tag must be located in the `templates` directory.

Encoding

All JavaScript files generated by the ClojureScript compiler must be UTF-8 encoded. The development server ensures that this happens by applying the `js-encoding` middleware.

Code reloading

Both Clojure and ClojureScript have a compile step. Having to actually execute a separate compile step while working on an application is not acceptable. If a source file changes, and we reload a page which uses that file, we need to see the change.

There are at least three kinds of the source files which can cause reloading: HTML, Clojure and ClojureScript.

If a ClojureScript file changes, all ClojureScript files will be reloaded. If an HTML template changes, all ClojureScript files will be reloaded. Reloading all files when one changes is a simple way to ensure that we will actually see the change. Reloading only the file that has changed will not do this in all cases. For example, if a macro or protocol is changed, dependent code will need to be reloaded.

A list of Clojure files to watch can be configured. If any file in the list changes, all of the files in the list will be reloaded. The list of files to watch is entered in the configuration map in the file `src/app/clj/one/sample/config.clj`. If `src/app/clj/one/sample/config.clj` is not itself in the list of files to watch, then you will have to restart the application when you change the list of watched files.

Configuration

The file, `src/app/clj/one/sample/config.clj`, provides a single place to configure many aspects of how the server works. This is where the list of Clojure files to watch is managed. It is also the place where the JavaScript which is added to the development and production views are configured.

In most cases, you will want the production view of the application to be different from the development view. At the very least, the ClojureScript One menu should be removed. The configuration map allows you to add a function under the `:prod-transform` key which will transform the host page in some way. In the sample application, the transformation will remove the menu items from the ClojureScript One menu.

The same transformation function is applied to the host page which is generated when building the artifacts which will be deployed.

Pasted from <https://github.com/brentonashworth/one/wiki/The-server>

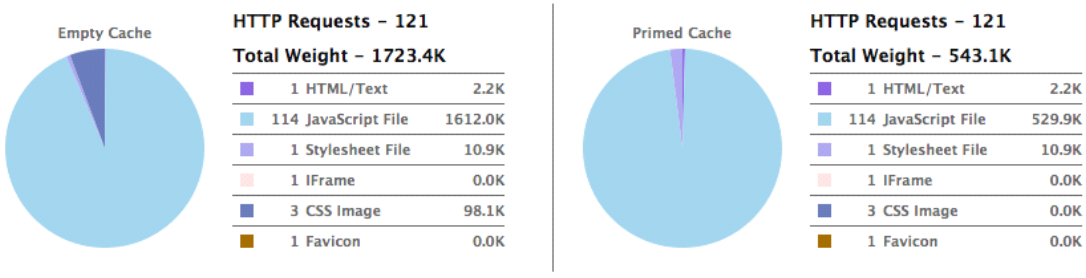
Production

Tuesday, February 14, 2012
8:59 AM

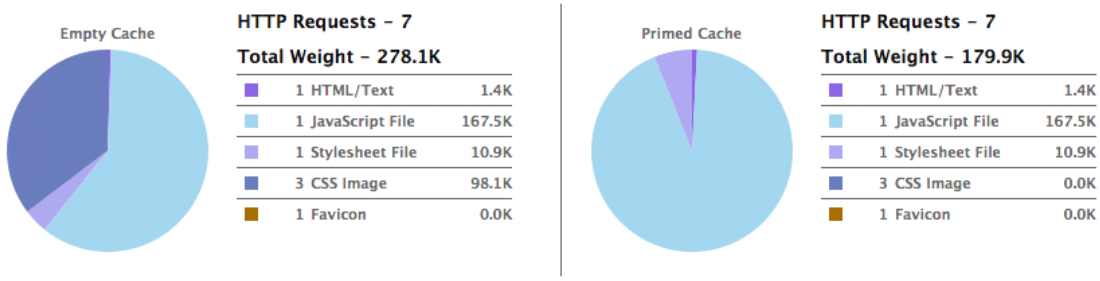
The ultimate goal of this project is to help you get an application into production.

ClojureScript utilizes Google Closure's compiler to compile all of the JavaScript code in your project into a single file. When you visit the development view of the application, each JavaScript file that is required as well as all of the compiled JavaScript for your application will be loaded as individual JavaScript files. For the sample application, at the time of this writing, this adds up to 114 JavaScript files and a total size of 1612K.

YSlow generates the following report:



When viewing the application in the production view, all JavaScript will exist in a single file. For the sample application, at the time of this writing, the file is 167.5K.



Apart from admiring how small the production version of the application is, the production view allows you to confirm that this version of the application works. It is possible to do things that are not compatible with advanced compilation.

One example of something that can break advanced compilation is evaluating a JavaScript string. This string may contain code which attempts to call a function using a name which will not exist after minification. The Google Closure compiler is not smart enough to know that the string contains code which must also be minified.

After making changes to the application, the production view may take a long time to load. This time is due to the time it takes to advanced compile the application.

Pasted from <https://github.com/brentonashworth/one/wiki/Production>

Building deployment artifacts

Tuesday, February 14, 2012
9:00 AM

Once the application is finished, you will want to create the artifacts which can be deployed. This includes the application's JavaScript compiled in advanced mode, the host HTML page and all of the resources in the public directory. Remember that the host page does not exist during development but is generated dynamically.

From the top level of your project, run the following commands.

```
lein run -m script.build
lein run -m script.serve
```

`lein run -m script.build` will generate all of the deployment artifacts in a directory named `out` at the top level of your project.`lein run -m script.serve` will run the production server which will serve this application as well as the backend API.

The produced `out` directory will have the basic structure shown below:

```
out
└── public
    ├── 404.html
    ├── css
    │   └── one.css
    ├── images
    │   └── favicon.ico
    ├── index.html
    └── javascripts
        └── mainp.js
```

The file `index.html` is the host page for the application and is generated when you run `lein run -m script.build`. If the application does not use a backend (it is pure JavaScript) then you can open the `index.html` file directly in a browser and everything will work. If the application does use a Clojure backend then you will need to start a server.

The sample application has a backend API and therefore a server is required for the application to work properly. A production server is implemented in the file `src/app/clj/one/sample/prod_server.clj` and can be run by executing `lein run -m script.serve`.

The `index.html` host file is generated using the same `application.html` template that was used during development. To transform this file for the production application, add a transformation function to the application configuration (`src/app/clj/one/sample/config.clj`) under the`:prod-transform` key. The sample application uses a transformation function which will remove the ClojureScript One navigation menu.

Pasted from <https://github.com/brentonashworth/one/wiki/Building-deployment-artifacts>

Deploying to Heroku

Tuesday, February 14, 2012
9:00 AM

You can deploy your One-based application wherever you like. If your application truly is standalone (i.e it has no server component), this will consist merely of copying the contents of the `out` directory somewhere. If, however, your application has a server component, you will need to host it somewhere that can both run the server and serve the rest of the application.

The included sample application does have a simple server component, and so needs somewhere to run. One easy way to deploy is to make use of Heroku, a hosting company that offers free hosting plans and easy deployment. We have tested deploying the sample application to Heroku using the following steps. You may need to alter these if your application differs significantly from the sample application.

Steps to Deploy

First, open a terminal and change to the ClojureScript One directory.

```
cd /path/to/one
```

If you're starting from a fresh checkout, then you may need to run `lein bootstrap` first.

Build the application, which compiles all the ClojureScript to JavaScript, and generates the HTML from the templates you've defined.

```
lein run -m script.build
```

Commit the newly generated content to the local git repository

```
git add out
git commit -m "Add build content"
```

If you have not already, sign up for a free account at Heroku and install the Heroku gem.

```
gem install heroku
```

Create a new Heroku application.

```
heroku apps:create pick-a-name --stack cedar
```

You will be prompted to enter a username and password for your Heroku account and to enable upload of your ssh public key. Do so.

Deploy your application to Heroku. Deployment happens automatically when you push your repository to Heroku via git.

```
git push heroku master
```

You should now be able to see your application running at `http://pick-a-name.herokuapp.com`.

Pasted from <https://github.com/brentonashworth/one/wiki/Deploy-to-Heroku>

How We Work

Tuesday, February 14, 2012
9:00 AM

This page is an overview of how ClojureScript One itself is developed. It documents how releases are named and performed and how contributions are integrated.

The purpose of ClojureScript One is to be an example of how best to leverage ClojureScript, and a big part of that is the documentation that ClojureScript One provides. Because of this, we have to be careful not to make changes that would cause the code and the documentation to get out of sync. The process described below will help us to accomplish this.

Releases and Milestones

ClojureScript One does not issue releases in the typical manner. That is, there is no "version 1.0.0" which is then followed by a "version 1.1.0", etc. Rather, we simply tag the master branch of the repository occasionally with something like "r166", which consists of a lower-case r followed by a number derived from git describe. The number will always increase with each successive "release", although it may increase by more than one.

Closely related to this are GitHub milestones. These are visible in the GitHub Issue Tracker, and have names like "M002", which is short for "milestone 2". Milestones are groups of related features that the ClojureScript One team is working towards releasing. They often share a common theme, and are generally tied to a date. If you want to know what the ClojureScript One team is planning on releasing next, just look at the next milestone.

Note that features that are assigned to no milestone may **never** be worked on. We put a lot of things into the issues list merely as a way to keep track of ideas that people have had. The presence of an issue in the list does not indicate our intention to implement. In contrast, assignment of an issue to a milestone indicates that we plan to work on the issue in the near future, and to release it along with the rest of the work in that milestone.

When it's time to release, we follow the procedures outlined in Release Checklist.

Git Branches

The master branch is intended to reflect the current, stable version of ClojureScript One. It should always be safe for people to check out this branch and work with the code in it. Further, it is this branch that the wiki and website describe.

Work against the next milestone takes place in a branch with the same name as the milestone. For example, work against the M002 milestone takes place in the M002 branch. These branches may not be stable, in that they may contain half-completed features, and may not correspond to the documentation in the wiki or on the website.

Similarly to the code, updates to the wiki will be maintained in a fork of the wiki itself. Only when master is updated with the latest release will the wiki be updated with documentation matching the changes in that release.

Contributing

ClojureScript One welcomes contributions from the community. However, because the master branch is intended to reflect the latest stable version of the project, it is not an appropriate target for pull requests. When sending pull requests, please target them at the pull-requests branch of the project. Pull requests will be merged there, and then merged into the appropriate milestone branch, where they will be tested and documented and eventually merged into master and released.

Pasted from <https://github.com/brentonashworth/one/wiki/HowWeWork>

Emacs

Tuesday, February 14, 2012
9:00 AM

Although you can use any editor you like to edit the ClojureScript One files, the authors (and many Clojure programmers) use emacs. On this page we document some of the things you can do to improve your experience working with ClojureScript One in emacs.

Using clojure-mode to edit .cljs files

By default, ClojureScript files are not associated with clojure-mode in emacs. Put the following lines into your `.emacs` or `.emacs.d/init.el` file:

```
(add-to-list 'auto-mode-alist '("\.cljs$" . clojure-mode))
```

Working with two REPLs

Working with ClojureScript One differs from working in Clojure in that you are often simultaneously working with two different REPLs: a ClojureScript REPL where you want to evaluate commands in the browser, and a Clojure REPL where you want to evaluate commands against the server portion of the application. A trick that we've used to pull this off is to run the one of them in an inferior lisp buffer, and the other in shell-mode. We define commands and keys for sending expressions to the shell buffer and use the existing commands to send expressions to the inferior lisp buffer. It's a bit of a hack, but an occasionally handy one.

To make it work, put the following into your `.emacs` or `.emacs.d/init.el` file:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Allow input to be sent to somewhere other than inferior-lisp
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; This is a total hack: we're hardcoding the name of the shell buffer
(defun shell-send-input (input)
 "Send INPUT into the *shell* buffer and leave it visible."
 (save-selected-window
  (switch-to-buffer-other-window "*shell*")
  (goto-char (point-max))
  (insert input)
  (comint-send-input)))

(defun defun-at-point ()
 "Return the text of the defun at point."
 (apply #'buffer-substring-no-properties
       (region-for-defun-at-point)))

(defun region-for-defun-at-point ()
 "Return the start and end position of defun at point."
 (save-excursion
  (save-match-data
   (end-of-defun)
   (let ((end (point)))
    (beginning-of-defun)
```

```
      (list (point) end)))))

(defun expression-preceding-point ()
 "Return the expression preceding point as a string."
 (buffer-substring-no-properties
  (save-excursion (backward-sexp) (point))
  (point)))

(defun shell-eval-last-expression ()
 "Send the expression preceding point to the *shell* buffer."
 (interactive)
 (shell-send-input (expression-preceding-point)))

(defun shell-eval-defun ()
 "Send the current toplevel expression to the *shell* buffer."
 (interactive)
 (shell-send-input (defun-at-point)))

(add-hook 'clojure-mode-hook
       '(lambda ()
         (define-key clojure-mode-map (kbd "C-c e") 'shell-eval-last-expression)
         (define-key clojure-mode-map (kbd "C-c x") 'shell-eval-defun)))
```

Once there, you can eval this code by selecting it and typing `M-x eval-region`. Any open Clojure files will need to be reopened for these changes to take effect.

For this example, we will start the Clojure REPL in the shell mode buffer and the ClojureScript REPL in the inferior lisp buffer. You can reverse these if it suits your preference.

- Run `M-x shell` to get a new `*shell*` buffer
- In the `*shell*` buffer, change to the One directory by typing `cd /path/to/one`
- Start a Clojure REPL by running `script/repl`
- Start the dev server in the REPL by executing `(use 'one.sample.dev-server)` and then `(run-server)`
- Open a new emacs window and switch to it: `C-x 2` followed by `C-x o`
- Change the default directory of the new window: `M-x cd` followed by `/path/to/one`
- Start another Clojure REPL by typing `C-u M-x inferior-lisp` followed by `script/repl`
- Start a ClojureScript REPL in the new Clojure REPL with the commands `(use 'one.sample.dev-server)` and `(cljs-repl)`
- Connect a browser session to the ClojureScript REPL by opening a browser and visiting http://localhost:8080/.

  You're now ready to run commands in either the Clojure REPL or the ClojureScript one. Use the following keys:

- `C-M-x`: Send the top-level expression at point to the inferior-lisp buffer (in our example, this is the ClojureScript REPL).
- `C-c C-e`: Send the expression that ends just before point to the inferior-lisp buffer (in our example, this is the ClojureScript REPL).

- `C-c x`: Send the top-level expression at point to the shell buffer (in our example, this is the Clojure REPL).
- `C-c e`: Send the expression that ends just before point to the shell buffer (in our example, this is the Clojure REPL).

  Consider the following code, with the cursor positioned at the location indicated by the `|` character:

  `(+ 2 | 3)`

  Here is what each of the above keys would result in:

- `C-M-x`: Evaluate `(+ 2 3)` in the ClojureScript REPL.
- `C-c C-e`: Evaluate `2` in the ClojureScript REPL.
- `C-c x`: Evaluate `(+ 2 3)` in the Clojure REPL.
- `C-c e`: Evaluate `2` in the Clojure REPL.

  If the keys don't work, it may be as a result of interference with slime minor mode. Try disabling slime-mode in your Clojure buffer.

  Pasted from <https://github.com/brentonashworth/one/wiki/Emacs>

Dependencies

Tuesday, February 14, 2012
9:00 AM

As mentioned elsewhere, this project depends on Clojure, Ring, Compojure, Enlive, ClojureScript and Domina. Running `lein deps` will get most of these dependencies, in the usual way, but ClojureScript and Domina are handled a bit differently. Running `lein bootstrap` will get all of the dependencies as described below.

The problem with releases

The software release process needs a bit of a reimagining. There are several problems with the status quo.

The first problem is that naming releases is hard. In the past we have struggled to impose some kind of meaning on the names that we give to releases of our software. The one common problem with all previous approaches is that, in the end, they are arbitrary. There is a disconnect between changes made to the source code and the name that the new version is given.

This disconnect points to another problem. It is difficult to audit changes between versions. Most of the time we are dependent on a change log which may or may not be accurate. Many businesses will not allow arbitrary binaries to be downloaded from the internet and used in a project without extensive testing and an audit of the changes that have been made. This is currently a very difficult process.

As the user of a library, it is difficult to experiment with the code in the context of the project where it is being used. This is the place where it makes the most sense to do experiments. Depending on the build process for a project, it can be difficult to make a change and then add the changed version as a dependency to our project. Although this can be done, it is usually just hard enough to prevent us from doing it.

Related to this is the difficulty in testing changes which library authors have made but which are not yet released. For software to maintain high quality, changes need to be tested by as many users as possible. Because of the barrier imposed by the release process, busy developers don't have the time to do this important job.

When using a library, if a problem is found, it is time consuming to fix. We have to make the change to the library and then depend on our changed version until the change is accepted into the main project and released.

Finally, we are dependent on the project authors to actually release the software on a schedule that we are happy with. From the author's perspective, they have to struggle with how best to release things so as to upset the least number of people.

Something must be done.

ClojureScript releases

ClojureScript has taken a non-traditional approach to releases.

There is only one source of truth for ClojureScript, the ClojureScript repository. This repository contains a complete history of every change that has ever been made to ClojureScript and who made each change.

Each commit made to this repository creates a new version of ClojureScript.

It is helpful to give names to versions. Names allow us to refer to and depend on specific versions. A Git repository gives each commit a unique name, the SHA. The problem with these names is that

they don't have an order. You can't tell from the names which version is newer than another version.

In ClojureScript, ordered version names are generated automatically without being arbitrary. The names are the number of commits on the master branch from the beginning of the project. Occasionally the repository will be tagged with a name like `r957`. This tag marks the 957th commit to this project.

What we want

For Clojure-based projects there are several things that we have come to expect from our build tools when managing dependencies:

1. flexible specification of dependent versions
2. installation on the classpath
3. portability
4. automatic retrieval of transitive dependencies

There are also many things we have not been able to easily do:

1. update library code in our running application
2. try our projects with changes made to a library
3. audit changes to a library
4. contribute changes back to a library

It would be nice if we could have it all.

Git Dependencies

ClojureScript One depends on ClojureScript and Domina as Git dependencies. The install process for ClojureScript One is

```
git clone git@github.com:brentonashworth/one.git
cd one
lein bootstap
```

`lein bootstrap` will get all dependencies for this project. Running `lein bootstrap` has the same effect as running the two commands:

```
lein deps
lein git-deps
```

ClojureScript and Domina are retrieved by cloning their git repositories into the `.lein-git-deps` directory and then checking out the version configured in `project.clj`.

The `project.clj` file for ClojureScript One shows how to configure Git dependencies.

```
:git-dependencies [["https://github.com/clojure/clojurescript.git"
                    "329708bdd0f039241b187bc639836d9997d8fbd4"]
                   ["https://github.com/levand/domina.git"
                    "c0eb06f677e0f9f72537682e3c702dd27b03e2e4"]]
```

A Git dependency is a URL for a repository and then, optionally, a tag, commit or branch. Directories that should be on the classpath have to be manually configured:

```
:extra-classpath-dirs [".lein-git-deps/clojurescript/src/clj"
                       ".lein-git-deps/clojurescript/src/cljs"
                       ".lein-git-deps/domina/src/cljs"
                       "src/app/cljs"
                       "src/app/cljs-macros"
                       "src/lib/clj"
                       "src/lib/cljs"
                       "templates"]
```

Notice that tasks like trying a branch, testing your project with the latest version of master, working from a fork and contributing changes back to the project are now trivial tasks supported by Git.

The one thing that this approach is missing is the ability to automatically calculate and retrieve transitive dependencies.

ClojureScript One will follow the same model as ClojureScript for releases. If you would like to depend on this project in your own projects then adding a Git dependency as we have done here is the recommended method.

Benefits

The benefits of working with version controlled source code instead of binaries are numerous. Think of the tools that could be created to automate things like testing to determine the newest version of the library that the project will work with or finding out the newest versions of each library that are compatible with each other.

Instead of a mysteriously named black box, every library becomes the source of a tremendous amount of information about the code that you depend on.

Maven

If you don't see any value in the above approach then you may add ClojureScript as a dependency in the usual way.

```
:dependencies [[org.clojure/clojurescript "0.0-927"]]
```

Domina and ClojureScript One will also have standard jar releases at some point in the future.

Pasted from <https://github.com/brentonashworth/one/wiki/Dependencies>